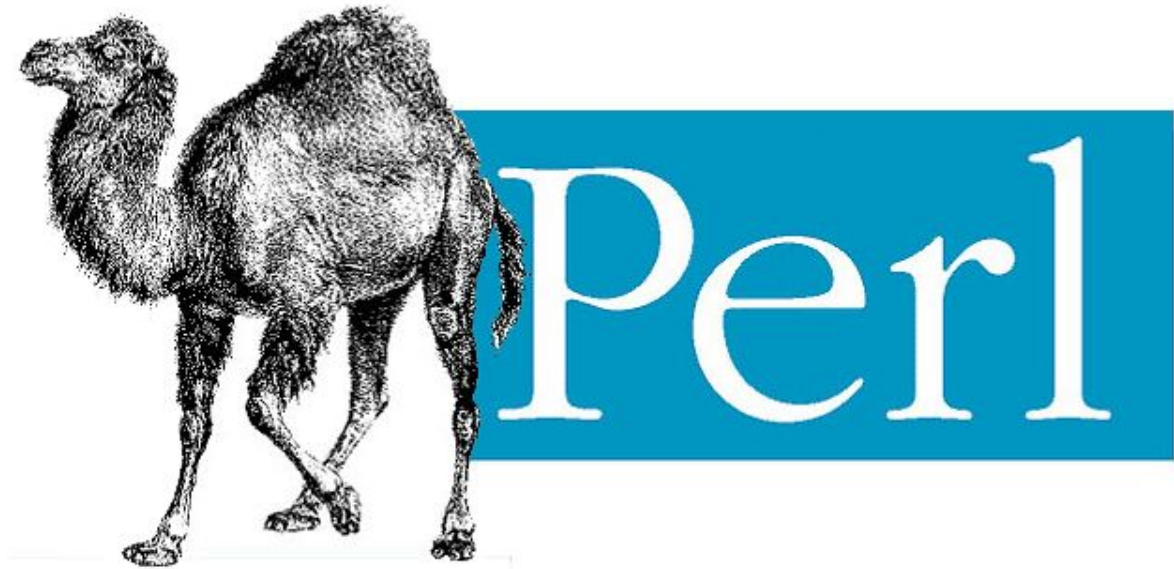


Linguagem Perl

Uma abordagem prática para Pentesters



por **Heitor Gouvêa**

Sumário

Sobre o autor	03
Introdução ao Perl	04
Ambiente de desenvolvimento	05
Olá Mundo	06
Executando nosso código	07
Variáveis	08
Operadores	10
Entrada de dados	11
Tomando decisões	14
Laços de repetição	19
Colorindo nosso script	21
Manipulação de arquivos e execução de comandos	23
Array	25
Foreach	26
ARGV's	27
Sub-rotinas	28

Sobre o autor

Heitor Gouvêa é programador Perl desde 2014, além de ser palestrante e escritor. Atua no campo da Segurança da Informação Ofensiva há mais de 3 anos, passando neste tempo por cargos operacionais como: Trainee, Analista de Segurança e Desenvolvimento, Consultor; E hoje atua como CTO da Inploit Security, empresa que fundou no início de 2018.

Autor da ferramenta Nipe, ferramenta responsável por garantir o anonimato a seus usuários, presente em várias distribuições Linux focadas em segurança da informação como: Black Arch, Weak Net e LionSec Linux.

Introdução ao Perl

Perl é uma linguagem de programação interpretada e de alto nível, usada para desenvolver aplicações web e desktop. Ela foi desenvolvida por Larry Wall em 1987, enquanto o mesmo trabalhava para o Laboratório de Propulsão de Jatos da NASA. A sigla PERL significa "Practical Extraction And Report Language" em português "Linguagem Prática de Relatório e Extração".

Perl se destaca por gratuita, de código aberto, rápida, multiparadigma, eficiente, segura, multiplataforma e de fácil manutenção.

Tal linguagem conseguiu reunir módulos, classes, scripts e frameworks desenvolvidos pela comunidade em um só lugar, chamado de CPAN (Comprehensive Perl Archive Network), um repositório onde você pode encontrar quase tudo já desenvolvido para a linguagem. Ela também se tornou muito popular fora do Brasil por ser uma linguagem que previne erros de segurança, é muito pouco provável que você cometa algum erro de segurança que comprometa sua aplicação.

Muitos Pentesters costumam usar Perl no dia-a-dia, por ela ser uma linguagem rápida, madura e super eficiente para trabalhar em situações comuns para a Web e para Redes.

Ambiente de desenvolvimento

Em nosso ambiente de desenvolvimento iremos precisar apenas do interpretador e de um editor de texto de sua preferência. Caso você use alguma distribuição Linux como o Ubuntu, Debian, Fedora ou Arch, você já possui um interpretador Perl instalado em sua máquina. Caso contrário, será necessário fazendo o download do mesmo.

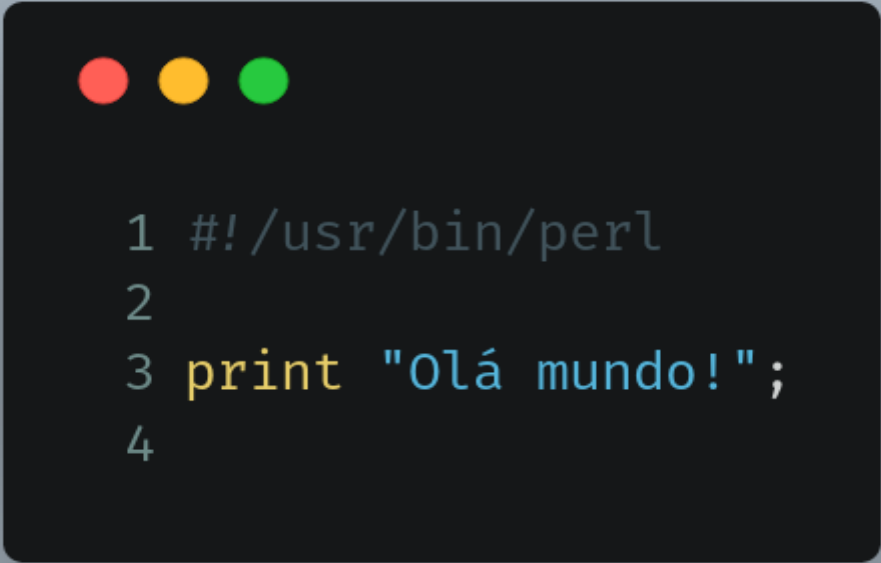
- Download do interpretador Perl

Você precisará instalar o “Active Perl” que pode ser adquirido em:

<http://activestate.com>

Olá mundo em Perl

Vamos escrever nosso primeiro programa em Perl agora. Abra o seu editor de texto e digite as seguintes instruções:



```
1 #!/usr/bin/perl
2
3 print "Olá mundo!";
4
```

A primeira linha, é uma instrução que diz ao seu sistema operacional que o programa que rodará é feito em Perl e, o mesmo já se encarrega de “chamar” o interpretador do Perl. Essa linha que escrevemos é para o Linux, mas fique tranquilo pois o próprio ActivePerl do Windows irá converter a mesma para o caminho do Windows.

Já na 3ª linha possuímos a instrução “**print**”, que significa que nosso programa irá escrever algo na tela, neste caso usamos as aspas duplas - porém, também podíamos usar aspas simples - para instruir que o que estiver dentro delas será a nossa mensagem exibida na tela, o ponto e vírgula sinaliza o final da instrução.

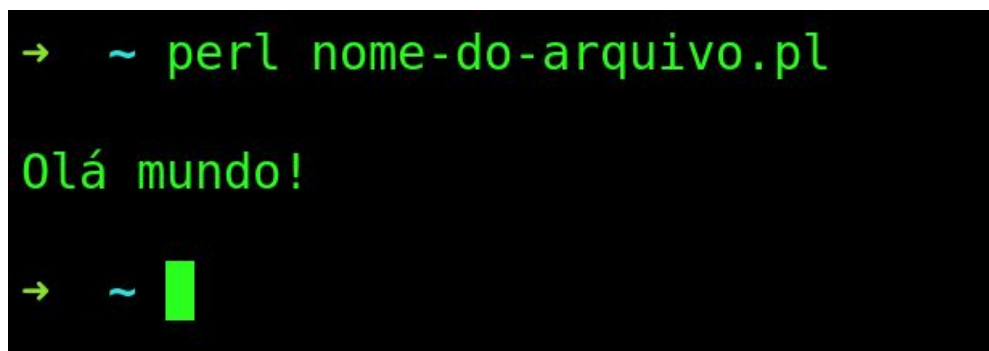
Executando nosso código

Todo código feito em Perl deve possuir a extensão de arquivo igual a “.pl”. Para executarmos o nosso primeiro código, basta salvar o arquivo, abrir o Terminal, ou o prompt de comando e digitar o seguinte comando:



A primeira instrução neste comando é responsável por executar o interpretador do Perl; Já o segundo argumento é responsável por transmitir para o interpretador o nome do arquivo que ele deve interpretar.

A nossa saída será:



Variáveis

Agora iremos fazer uma introdução a variáveis. Afinal, o que é uma variável? Na programação uma variável é um “espaço” capaz de armazenar e representar um valor ou expressão e este valor ou expressão pode variar, ou seja, mudar durante o decorrer do tempo.

Em Perl uma variável pode ser declarada da seguinte forma:



```
1 #!/usr/bin/perl
2
3 print "Olá mundo!";
4
5 my $variavel = "ola";
```

Toda variável possui um **\$** antecedendo seu nome, isto é obrigatório. Neste caso da imagem o valor da nossa variável será uma string, tudo que estiver dentro de aspas duplas ou simples é o conteúdo da variável, o ponto e vírgula, como dito antes indica o final do comando.

Em Perl as variáveis são dinamicamente tipadas, ou seja, você não precisa definir o tipo de dado que uma variável irá suportar antes de usar.

Agora que sabemos o que é uma variável, podemos realizar muitas outras coisas.

Vamos dar início a um simples programa que some alguns números e

escreva o resultado na tela.

```
#!/usr/bin/perl

my $numero = "10";
my $numero2 = "20";

print "\nPrimeiro valor $numero. Segundo valor $numero2\n";

my $soma = $numero + $numero2;

print "\nA soma total entre eles é de $soma\n";

exit;
```

Podemos ver que definimos duas variáveis, **\$numero** que é igual a 10 e **\$numero2** que é igual a 20, logo depois escrevemos isto na tela, e na linha 8 realizamos a soma das 2 variáveis que resultou no valor de 30.

O comando “\n” indica uma quebra de linha, ele fará que o conteúdo que vier depois seja escrito na próxima linha.

A saída do programa ficará assim:


```
→ ~ perl soma.pl

Primeiro valor 10. Segundo valor 20

A soma total entre eles é de 30
→ ~ █
```

Operadores

Perl usa todos os mesmos operadores da linguagem C:

```
$a = 1 + 2;      # Soma 1 em 2 e armazena o resultado em $a
$a = 3 - 4;      # Subtrai 4 de 3 e armazena o resultado em $a
$a = 5 * 6;      # Multiplica 5 por 6
$a = 7 / 8;      # Divide 7 por 8 e obtém 0,875
$a = 9 ** 10;    # Eleva nove a décima potência
$a = 5 % 2;      # Armazena em $a o resto da divisão de 5 por 2
++$a;           # Incrementa $a e depois retorna
$a++;           # Retorna $a e depois incrementa
--$a;           # Decrementa $a e depois retorna
$a--;           # Retorna $a e depois decrementa

# Para strings, estas são algumas das maneiras:

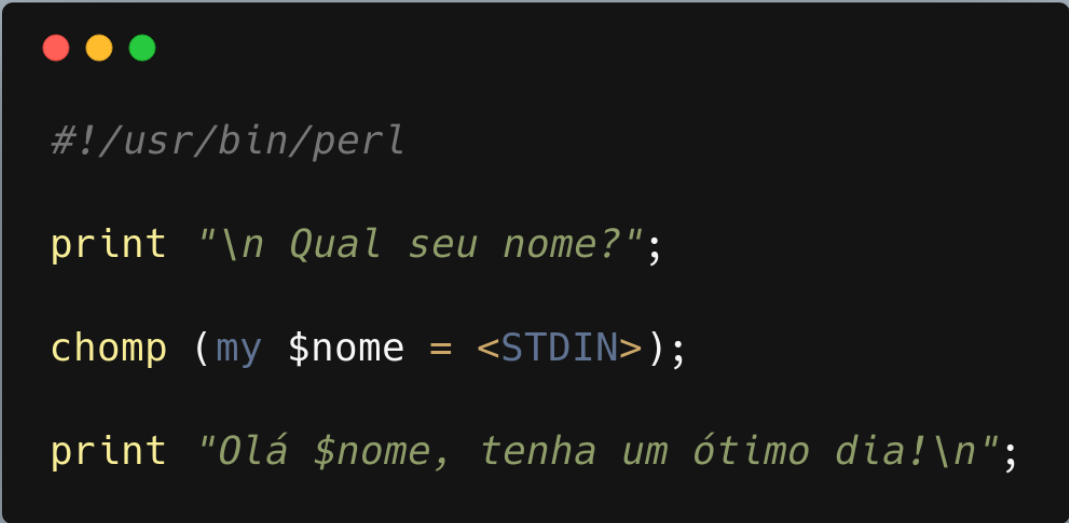
$a = $b . $c;    # Concatena $b com $c
$a = $b x $c;    # $b é repetida $c vezes

# Para designar valores temos as seguintes formas:

$a = $b;         # Coloca em $a o conteúdo de $b
$a += $b;        # Soma o valor de $b ao valor de $a
$a -= $b;        # Subtrai o valor de $b do valor de $a
$a .= $b;        # Concatena $b a $a
```

Entrada de dados

Agora que já temos um conhecimento sólido sobre variáveis, iremos fazer que o usuário defina o conteúdo da variável.



```
#!/usr/bin/perl

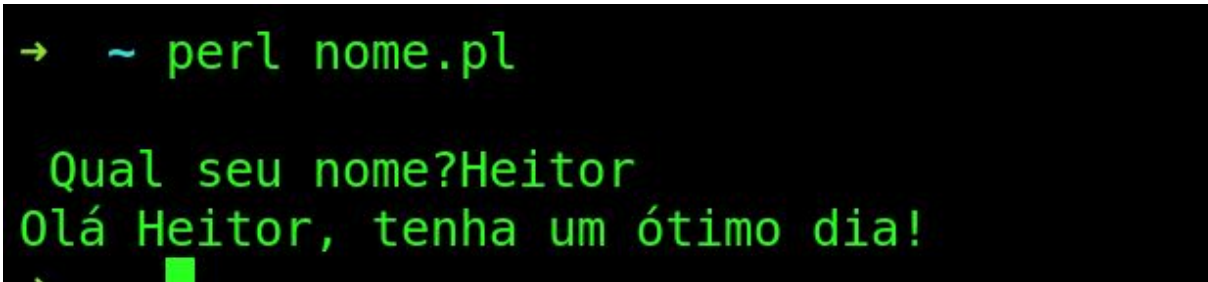
print "\n Qual seu nome?";

chomp (my $nome = <STDIN>);

print "Olá $nome, tenha um ótimo dia!\n";
```

Na linha 4 nós declaramos a variável **\$nome** que é igual á **<STDIN>** , o **<STDIN>** é a função que lê uma linha de texto gerada pelo teclado. “E o que é **chomp?**” - é a função que elimina o último caracter se caso ele for um "fim de linha" (caracter **\n**)...

A saída deverá ser assim:



```
→ ~ perl nome.pl

Qual seu nome?Heitor
Olá Heitor, tenha um ótimo dia!
```

Considerando o conhecimento que temos até o momento, é hora de colocá-lo em prática para descobrir do que já somos capazes de fazer!

```
#!/usr/bin/perl

use strict;
use warnings;

print "\nQual é o seu nome?";
chomp (my $nome = <STDIN>);

print "\nOlá $nome, pode me dizer a sua idade?";
chomp (my $idade = <STDIN>);

my $nascimento = 2018 - $idade;

print "Você provavelmente nasceu no ano $nascimento";

exit;
```

Bom, temos algumas coisinhas novas... Primeiramente vamos entender o que são módulos! Módulos são praticamente bibliotecas que mudam o modo de interpretação do Perl. Como usar módulos? Na 3ª e 4ª linha nós usamos os módulos **'strict'** e **'warnings'**. O comando **'use'** ordena ao Perl que ele carregue e ative cada um dos módulos. Os módulos **strict** e **warnings** irão ajudá-lo a capturar alguns erros e enganos comuns em seu código ou até mesmo em alguns casos prevenir que você os realize. Ambos são extremamente úteis, nunca deixe de usá-los.

Logo depois na linha 12 efetuamos uma operação matemática para descobrir o ano de nascimento do nosso usuário baseado na idade informada, a lógica para resolver esse problema é basicamente:

ano atual, menos (-) a idade e isso resulta no ano de nascimento do usuário

Depois escrevemos isto na tela utilizando o comando **print**. A saída deverá ocorrer assim:

```
→ ~ perl teste.pl
```

```
Qual é o seu nome?Heitor
```

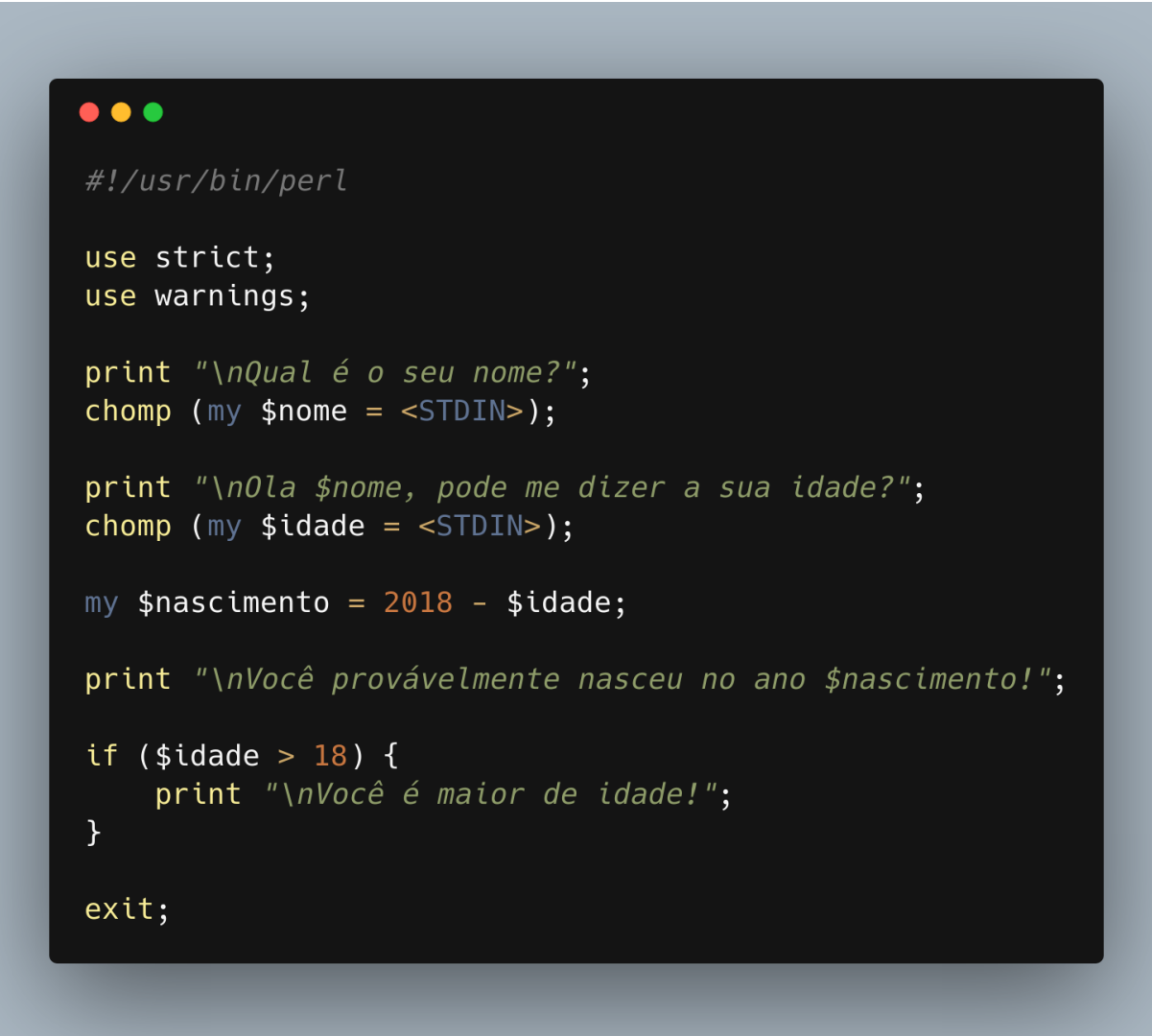
```
Olá Heitor, pode me dizer a sua idade?19
```

```
Você provavelmente nasceu no ano 1999
```

Tomando decisões

Agora que já conseguimos fazer com que o usuário nos forneça alguns dados, vamos fazer o nosso programa tomar decisões. Para isso o Perl possui os comparadores “**if**”, “**else**” e “**elsif**”.

Primeiro vamos entender o **if** e logo depois iremos para os próximos. O comparador **if** traduzido para o português significa “se”. Um bom exemplo é a imagem abaixo:



```
#!/usr/bin/perl

use strict;
use warnings;

print "\nQual é o seu nome?";
chomp (my $nome = <STDIN>);

print "\nOla $nome, pode me dizer a sua idade?";
chomp (my $idade = <STDIN>);

my $nascimento = 2018 - $idade;

print "\nVocê provavelmente nasceu no ano $nascimento!";

if ($idade > 18) {
    print "\nVocê é maior de idade!";
}

exit;
```

Na linha 16 podemos ver o **if** em ação, neste caso ele compara se a variável que contém a idade informada pelo usuário é menor que o valor **18**, se a idade for maior que **18**, nós escrevemos com o comando **print** a seguinte mensagem:

“Você é maior de idade”.

A saída deverá ficar assim:

```
→ ~ perl idade.pl  
  
Qual é o seu nome?Heitor  
  
Ola Heitor, pode me dizer a sua idade?19  
  
Você provavelmente nasceu no ano 1999!  
Você é maior de idade!##
```

Com o **if** podemos comparar tanto valores numéricos e strings. Agora que entendemos o **if** podemos ir para o **else**: ele é como se fosse um “**se não**”, vamos ao exemplo:

```
#!/usr/bin/perl

use strict;
use warnings;

print "\nQual é o seu nome?";
chomp (my $nome = <STDIN>);

print "\nOla $nome, pode me dizer a sua idade?";
chomp (my $idade = <STDIN>);

my $nascimento = 2018 - $idade;

print "\nVocê provavelmente nasceu no ano $nascimento!";

if ($idade > 18) {
    print "\nVocê é maior de idade!";
}

else {
    print "\nVocê é menor de idade!";
}

exit;
```

Logicamente se a idade do usuário não for menor que 18, então ele será maior de idade.

```
→ ~ perl idade.pl
```

```
Qual é o seu nome?Heitor
```

```
Ola Heitor, pode me dizer a sua idade?17
```

```
Você provavelmente nasceu no ano 2001!
```

```
Você é menor de idade!##
```


Agora é hora do **elsif**: praticamente seria um “ou se não”. Com o **if**, **elsif** e **else** podemos tomar decisões muito mais precisas. Vamos ao exemplo:

```
#!/usr/bin/perl

use strict;
use warnings;

print "\nVocê gosta de bolachas?";
chomp (my $resposta = <STDIN>);

if ($resposta eq "sim") {
    print "\nInteressante! Também gosto!";
}

elsif ($resposta eq "não") {
    print "\nQue pena, iria te oferecer algumas!";
}

else {
    print "\nResponda com: 'sim' ou 'não'!";
}

exit;
```

Percebeu que temos coisas novas?

Conjuntos de operadores de comparação

O Perl possui dois conjuntos de operadores de comparação.

Numérico	String	Significando
==	eq	igual
!=	ne	diferente
<	lt	menor que
>	gt	maior que
<=	le	menor ou igual a
>=	ge	maior ou igual a

Na linha 9 vimos se a resposta era igual a “sim”, na linha 13 vimos se era igual á “não”, caso não fosse igual a nenhum dos dois, usamos o **else** para pedir para responder corretamente.

Saída:

```
→ ~ perl teste.pl
Você gosta de bolachas?sim
Interessante! Também gosto!##
→ ~ perl teste.pl
Você gosta de bolachas?não
Que pena, iria te oferecer algumas!##
→ ~ perl teste.pl
Você gosta de bolachas?talvez
Responda com: 'sim' ou 'não'!##
```

Laços de repetição.

Em Perl possuímos o comando “**while**” que em português significa “**enquanto**”, com ele podemos criar laços de repetição. Vamos usar como base o exemplo abaixo:

```
#!/usr/bin/perl

use strict;
use warnings;

my $unidade = "0";

print "\nQuantas vezes é necessário escrever 'Bom dia!' na tela?";
chomp (my $quantidade = <STDIN>);

while ($unidade <= $quantidade) {
    print "Bom dia\n";
    $unidade++;
}

exit;
```

Na 6ª linha definimos a variável **\$unidade** que é igual a 0, logo depois pedimos para o usuário informar uma quantidade que ele achar melhor, então na 11ª linha criamos a seguinte lógica:

Enquanto \$unidade for menor ou igual a \$quantidade vamos escrever ‘Bom dia’ na tela e retornar \$unidade e depois a incrementar.

Saída:

```
→ ~ perl enquanto.pl

Quantas vezes é necessário escrever 'Bom dia!' na tela?5
Bom dia
Bom dia
Bom dia
Bom dia
Bom dia
Bom dia
```

Simple né? Pois podemos utilizar o **for** para a mesma finalidade, porém em situações diferentes siga o exemplo:

```
#!/usr/bin/perl

use strict;
use warnings;

print "\nQuantas vezes é necessário escrever 'Boa noite!' na tela? ";
chomp (my $quantidade = <STDIN>);

for (my $unidade = 0; $unidade <= $quantidade; $unidade++) {
    print "\nBoa noite!";
}

exit;
```

While e **For** possuem praticamente a mesma finalidade, mas você deve saber quando usar cada um deles. A saída desses dois casos serão exatamente iguais.

Colorindo nosso script

Que tal deixarmos nossos programas um tanto mais colorido? Para isso faremos uso do módulo “**Term::ANSIColor**” - caso você utilize Windows, use o módulo “**Win32::Console::ANSI**”.


Provavelmente você não tem este módulo em seu sistema, para instalar digite o comando abaixo no terminal/prompt:

cpan install Term::ANSIColor

Em sistemas windows use:

cpan install Win32::Console::ANSI

A instalação de qualquer módulo segue o mesmo padrão, o comando “**cpan**” chama o nosso amiguinho cpan e o “**install**” avisa que queremos instalar um módulo, logo após colocamos o nome do módulo, neste caso foi o “**Term::ANSIColor**”. Após a instalação, vamos colocar a mão na massa!



```
#!/usr/bin/perl

use strict;
use warnings;
use Term::ANSIColor;

print "\nQuantas vezes é necessário escrever 'Bom dia' na tela?";
chomp (my $quantidade = <STDIN>);

for (my $unidade = 0; $unidade <= $quantidade; $unidade++) {
    print color("red"), "\nBom dia";
}

print color("green"), "\nEscrevi $quantidade vezes 'Bom dia'";

exit;
```

Acredito que este módulo não necessite de muita explicação mas: na 5ª linha nós o carregamos e o ativamos, logo nas próximas linhas com o comando **'print'** colocamos a cor com a opção:

color ("nome");


Os nomes das cores sempre deverão ser escrito em inglês e o resultado será assim:

```
→ ~ perl color.pl  
Quantas vezes é necessário escrever 'Bom dia' na tela?3  
Bom dia  
Bom dia  
Bom dia  
Bom dia  
Escrevi 3 vezes 'Bom dia'##
```

Manipulação de arquivos e comandos no sistema

Manipular arquivos e executar comandos no sistema fica muito fácil com Perl, podemos criar, editar e excluir arquivos. No exemplo abaixo podemos reparar que fizemos uso do comando:

system ("comando do sistema")



```
#!/usr/bin/perl

use strict;
use warnings;

print "\nNome do diretório: ";
chomp (my $dir = <STDIN>);
system ("mkdir $dir");

print "\nNome do arquivo de texto ";
chomp (my $arquivo = <STDIN>);
open(my $file, ">", "$arquivo");

print "\nConteúdo do arquivo: ";
chomp (my $content = <STDIN>);
print $file $content;

close($file);

exit;
```

O comando “**system**” é responsável por avisar que o conteúdo entre parênteses e aspas será executado diretamente no sistema, sendo assim os comandos podem variar de sistema operacional para sistema operacional. O comando “**mkdir**” é um comando do sistema Linux que é responsável por criar diretórios.

Fizemos um pedido para que o usuário fornecesse o nome do diretório e depois nós o criamos, logo nas próximas linhas pedimos o nome do arquivo e também o criamos caso já não exista, escrevemos o conteúdo que o usuário define no arquivo.

Saída:

```
→ perl ls -al
total 12
drwxr-xr-x  2 root root 4096 jun 12 19:22 .
drwxr-xr-x 48 root root 4096 jun 12 19:22 ..
-rw-r--r--  1 root root  336 jun 12 19:22 file.pl
→ perl perl file.pl

Nome do diretório:teste-diretorio


Nome do arquivo de texto:meu-arquivo.txt

Conteúdo do arquivo:conteudo do arquivo
→ perl ls -al
total 20
drwxr-xr-x  3 root root 4096 jun 12 19:23 .
drwxr-xr-x 48 root root 4096 jun 12 19:23 ..
-rw-r--r--  1 root root  336 jun 12 19:22 file.pl
-rw-r--r--  1 root root   19 jun 12 19:23 meu-arquivo.txt
drwxr-xr-x  2 root root 4096 jun 12 19:23 teste-diretorio
→ perl cat meu-arquivo.txt
conteudo do arquivo#
```

- \$arquivo: abre ARQUIVO apenas para leitura (o mesmo que <\$arquivo);
- >\$arquivo: abre ARQUIVO para escrita, criando-o caso não exista;
- >>\$arquivo abre ARQUIVO para modificação (append);
- +>\$arquivo: abre ARQUIVO para leitura/escrita.

Array

Um Array é quase mesma coisa que uma variável, porém um Array é capaz de armazenar vários itens, isso pode ser muito útil. Exemplo:

A terminal window with a dark background and three colored window control buttons (red, yellow, green) in the top left corner. The code is written in a light green font.

```
#!/usr/bin/perl

use strict;
use warnings;

my @frutas = ( "Uva", "Banana", "Laranja");
print $frutas[2];

exit;
```

Saída:

A terminal window showing the execution of the Perl script. The command is entered in green text, and the output is also in green text.

```
→ ~ perl array.pl

Laranja#
```

Na linha 6 declaramos o **array** da seguinte maneira:

```
my @frutas = ("Uva", "Banana", "Laranja");
```

Em seguida escrevemos na tela o 2º item, sendo ele a goiaba. Um array se conta da seguinte forma: 0 - 1 - 2 - 3 ... 1º Elemento = 0 e assim por diante.

Foreach

Junto com o array nós teremos mais uma opção, o “**foreach**” . O **foreach** é um comando que é responsável por percorrer os valores de um **array**; usando ele podemos fazer o que quisermos com todos os elementos de um **array** sem precisar ‘chamar’ um por um, por exemplo:

A screenshot of a code editor window with a dark background and light-colored text. The code is a Perl script that uses the 'foreach' loop to iterate over an array of fruits. The script starts with a shebang line, followed by 'use strict;' and 'use warnings;'. It then defines an array '@frutas' with the values 'Uva', 'Banana', and 'Laranja'. A 'foreach' loop iterates over each element in the array, printing a message for each fruit. The script ends with 'exit;'.

```
#!/usr/bin/perl

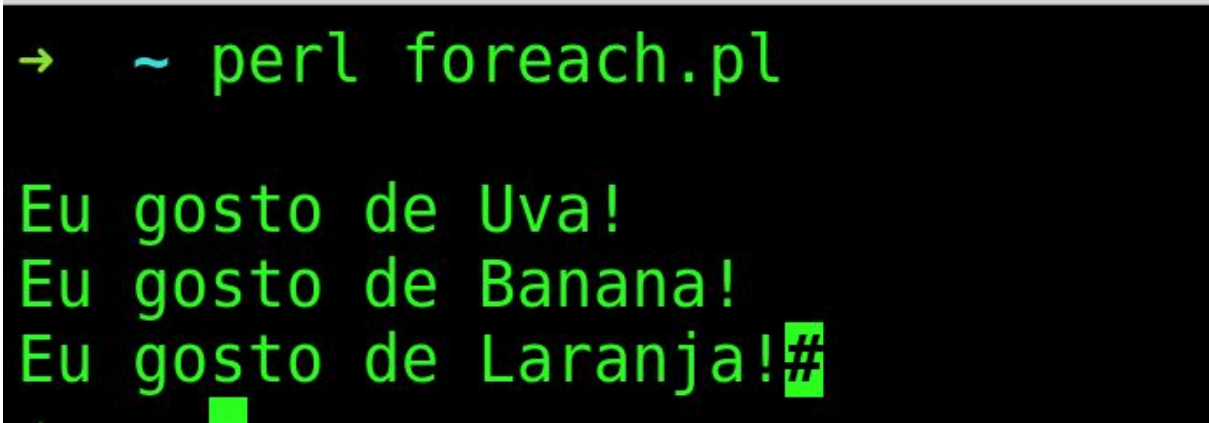
use strict;
use warnings;

my @frutas = ( "Uva", "Banana", "Laranja" );

foreach my $fruta (@frutas) {
    print "\nEu gosto de $fruta!";
}

exit;
```

Saída:

A screenshot of a terminal window with a black background and green text. It shows the command to run the Perl script and its output. The command is 'perl foreach.pl'. The output consists of three lines, each corresponding to a fruit in the array: 'Eu gosto de Uva!', 'Eu gosto de Banana!', and 'Eu gosto de Laranja!'. The prompt character is a green arrow pointing right.

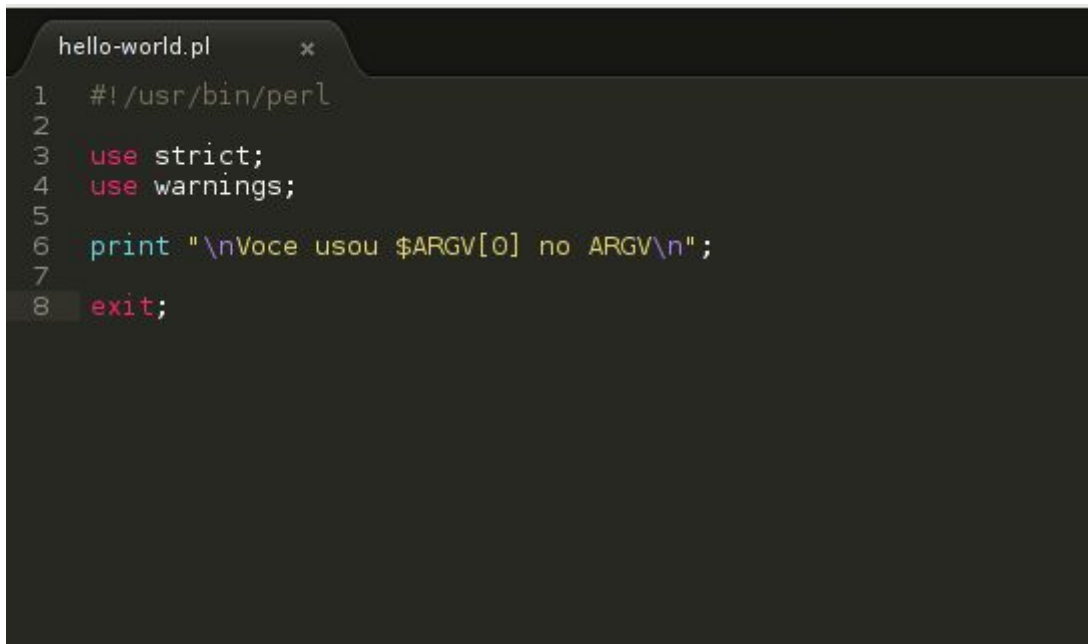
```
→ ~ perl foreach.pl

Eu gosto de Uva!
Eu gosto de Banana!
Eu gosto de Laranja!#
```

ARGV's

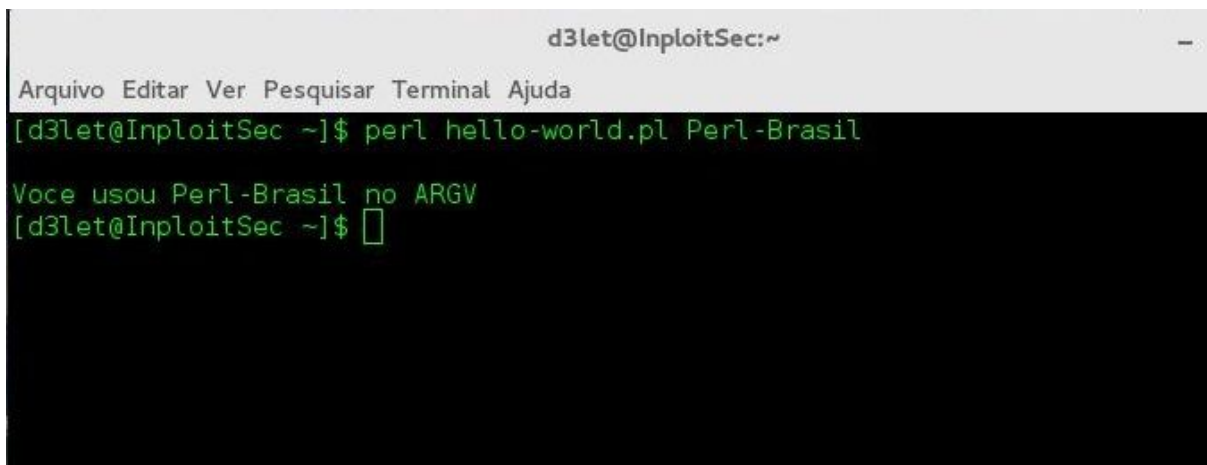
Aprenderemos agora um pouco sobre ARGV's, bom, ARGV's são simplesmente alguns parâmetros que têm acesso a linha de comando no qual nosso Script está sendo executado. Até aqui sempre executamos nossos códigos com o comando "perl hello-world.pl", mas nós poderíamos iniciar ele com outras coisas ao final do .pl por exemplo "perl hello-word.pl start" e pra isso teríamos que usar ARGV's.

Vamos a um exemplo:

A screenshot of a code editor window titled 'hello-world.pl'. The code is written in Perl and consists of eight lines: 1. #!/usr/bin/perl, 2. (empty), 3. use strict;, 4. use warnings;, 5. (empty), 6. print "\nVoce usou \$ARGV[0] no ARGV\n";, 7. (empty), 8. exit;.

```
1 #!/usr/bin/perl
2
3 use strict;
4 use warnings;
5
6 print "\nVoce usou $ARGV[0] no ARGV\n";
7
8 exit;
```

Saída:

A screenshot of a terminal window with the title 'd3let@InploitSec:~'. The terminal shows the command 'perl hello-world.pl Perl-Brasil' being executed, which results in the output 'Voce usou Perl-Brasil no ARGV'. The prompt '[d3let@InploitSec ~]\$' is shown again at the end of the line.

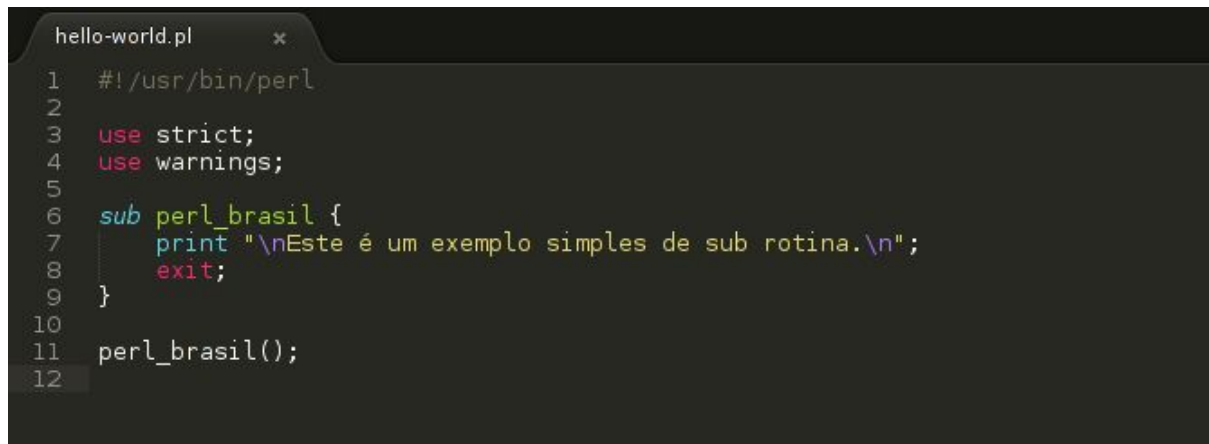
```
d3let@InploitSec:~
Arquivo Editar Ver Pesquisar Terminal Ajuda
[d3let@InploitSec ~]$ perl hello-world.pl Perl-Brasil
Voce usou Perl-Brasil no ARGV
[d3let@InploitSec ~]$
```

ARGV é um assunto até bem simples, e muito útil.

Sub-rotinas


Sub-rotinas são uma espécie de “função” que são apenas subs caminhos que seu programa pode tomar a qualquer momento ou local de seu programa, elas servem para deixar o código mais organizado, resolver alguns problemas específicos que podem ocorrer múltiplas ou não.

Exemplo:



```
hello-world.pl x
1  #!/usr/bin/perl
2
3  use strict;
4  use warnings;
5
6  sub perl_brasil {
7      print "\nEste é um exemplo simples de sub rotina.\n";
8      exit;
9  }
10
11 perl_brasil();
12
```

Saída:



```
d3let@InploitSec:~
Arquivo Editar Ver Pesquisar Terminal Ajuda
[d3let@InploitSec ~]$ perl hello-world.pl Perl-Brasil
Este é um exemplo simples de sub rotina.
[d3let@InploitSec ~]$
```

Agora que já entendemos como as sub rotinas funcionam, vamos a um exemplo mais complexo para reforçarmos nosso conhecimento sobre o mesmo: